



EECS E6893 Big Data Analytics

HW2: Data Analytics Pipeline using Airflow

TUTORIAL 2

October 2, 2026

Agenda

- Data pipeline orchestration using Apache Airflow

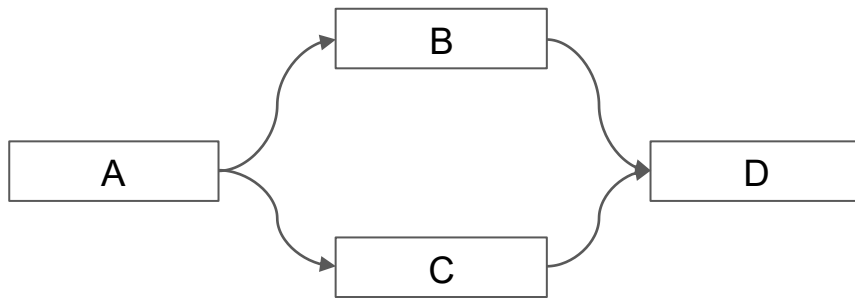
Airflow

Workflow

- A sequence of tasks involved in moving from the beginning to the end of a working process
- Started on a schedule or triggered by an event

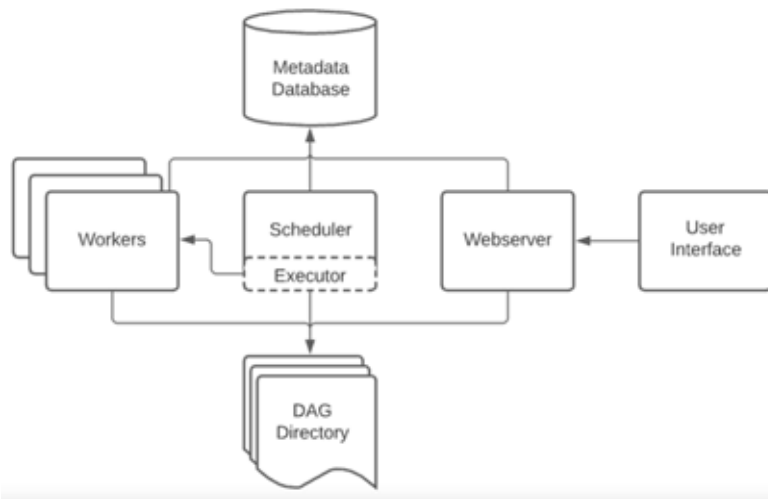
DAG (Directed Acyclic Graph)

- In Airflow, workflows are created using DAGs
- A DAG is a collection of tasks that you want to schedule and run, organized in a way that reflects their relationships and dependencies
- The tasks describe what to do, e.g., fetching data, running analysis, triggering other systems, or more
- A DAG ensures that each task is executed at the right time, in the right order, or with the right issue handling
- A DAG is written in Python



Airflow architecture

- **Scheduler:** handles both triggering scheduled workflows, and submitting tasks to the executor to run.
- **Executor:** handles running tasks.
- **Webserver:** a handy user interface to inspect, trigger and debug the behavior of DAGs and tasks.
- **A folder of DAG files:** read by the scheduler and executor
- **A metadata database:** used by the scheduler, executor and webserver to store state

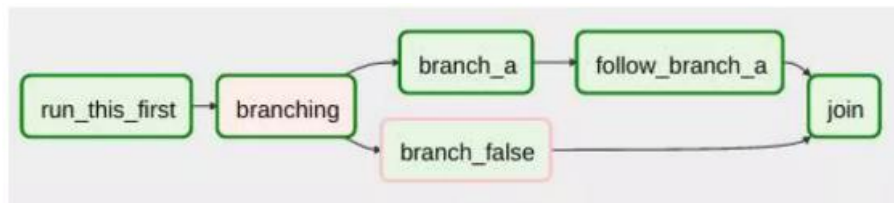


Executors

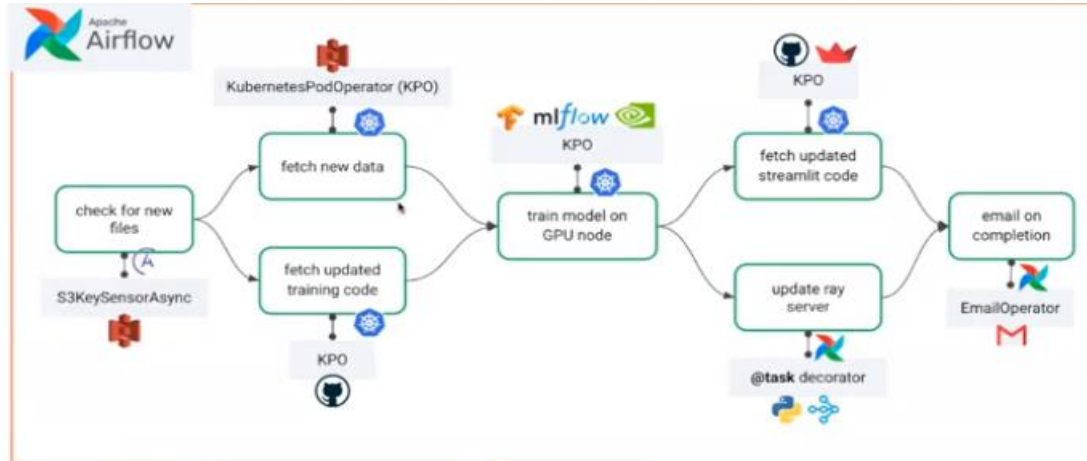
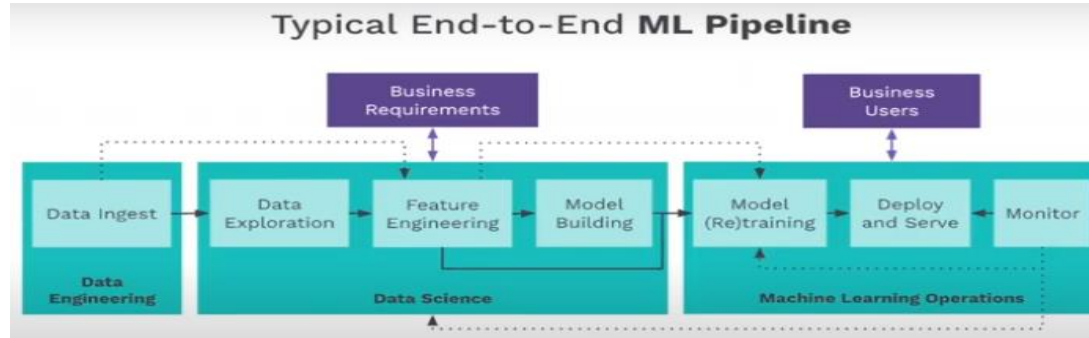
- **SequentialExecutor:** The Sequential Executor in Apache Airflow executes tasks one after the other in the order they are scheduled. While simple and easy to set up, it lacks parallelism and doesn't support concurrent task execution, making it suitable only for very basic, non-concurrent workflows and small-scale development setups.
- **LocalExecutor:** The Local Executor, unlike the Sequential Executor, allows limited parallelism by running tasks in separate processes on the same machine.
- **CeleryExecutor:** The Celery Executor in Apache Airflow leverages Celery, a distributed task queue system, to enable concurrent and parallel task execution across multiple worker nodes.
- **KubernetesExecutor:** The Kubernetes Executor in Airflow leverages Kubernetes clusters to distribute task execution across containers, allowing seamless scaling of resources based on demand.

Airflow vs Streaming solutions

- Airflow is not a streaming solution. Tasks do not move data from one to another in a dynamic fashion as we do in streaming solutions.
- Workflows are expected to be slightly more static or slowly changing. This allows for clarity around a unit of work and consistent orchestration cycles.



Sample Use Case using Airflow for ML Pipelines



HW CLARIFICATIONS:

Task 2 Build workflows

Q2.1 Implement this DAG (20 pts)

- Tasks and dependencies (5 pts)
- Manually trigger it (10 pts)
- Schedule the first run immediately and running the program every 30 minutes (5 pts)



Tasks and Scheduler

```
from datetime import datetime, timedelta
from textwrap import dedent
import time

# The DAG object; we'll need this to instantiate a DAG
from airflow import DAG

# Operators; we need this to operate!
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator

# These args will get passed on to each operator
# You can override them on a per-task basis during operator initialization
default_args = {
    'owner': 'cong',
    'depends_on_past': False,
    'email': ['ch321@columbia.edu'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(seconds=30),
    # 'queue': 'bash_queue',
    # 'pool': 'backfill',
    # 'priority_weight': 10,
    # 'end_date': datetime(2018, 1, 1),
    # 'wait_for_downstream': False,
    # 'dag': dag,
    # 'sla': timedelta(hours=2),
    # 'execution_timeout': timedelta(seconds=300),
    # 'on_failure_callback': some_function,
    # 'on_success_callback': some_other_function,
    # 'on_retry_callback': another_function,
    # 'sla_miss_callback': yet_another_function,
    # 'trigger_rule': 'all_success'
}
```

```
with DAG(
    'helloworld',
    default_args=default_args,
    description='A simple toy DAG',
    schedule_interval=timedelta(days=1),
    start_date=datetime(2021, 1, 1),
    catchup=False,
    tags=['example'],
) as dag:
```

```
def correct_sleeping_function():
    """This is a function that will run within the DAG execution"""
    time.sleep(2)
```

```
with DAG(
    'helloworld',
    default_args=default_args,
    description='A simple toy DAG',
    schedule_interval=timedelta(days=1),
    start_date=datetime(2021, 1, 1),
    catchup=False,
    tags=['example'],
) as dag:

    # 4 examples of tasks created by instantiating operators
    t1 = PythonOperator(
        task_id='t1',
        python_callable=correct_sleeping_function,
    )

    t2_1 = PythonOperator(
        task_id='t2_1',
        python_callable=correct_sleeping_function,
    )

    t2_2 = PythonOperator(
        task_id='t2_2',
        python_callable=correct_sleeping_function,
        retries=3,
    )

    t2_3 = PythonOperator(
        task_id='t2_3',
        python_callable=correct_sleeping_function,
    )

    t3_1 = PythonOperator(
        task_id='t3_1',
        python_callable=correct_sleeping_function,
    )

    t3_2 = PythonOperator(
        task_id='t3_2',
        python_callable=correct_sleeping_function,
    )

    t4_1 = BashOperator(
        task_id='t4_1',
        bash_command='sleep 2',
        retries=3,
    )
```

Operators

1. **PythonOperator**
2. **BashOperator**
3. branch_operator
4. email_operator
5. mysql_operator
6. DataprocOperator

...

...

PythonOperator:

```
def function():  
    print(123)  
  
task = PythonOperator(  
    task_id='task_id',  
    python_callable=function,  
)
```

BashOperator:

```
task = BashOperator(  
    task_id='task_id',  
    bash_command='sleep 2',  
)  
# other examples  
bash_command='python python_code.py'  
bash_command='bash bash_code.sh '  
                (must have a space to  
                satisfy Jinja template !!)
```

Operators – Suggested Examples

Python Operators

```
count = 0

def correct_sleeping_function():
    """This is a function that will run within the DAG execution"""
    time.sleep(2)

def count_function():
    global count
    count += 1
    print('output of count_increase: {}'.format(count))
    time.sleep(2)

def print_function():
    print('This represents a python operator')
    time.sleep(2)
```

```
t7 = PythonOperator(
    task_id='t7',
    python_callable=correct_sleeping_function,
)

t8 = PythonOperator(
    task_id='t8',
    python_callable=print_function,
)
```

Bash Operators

```
t5 = BashOperator(
    task_id='t5',
    bash_command='sleep 1',
)

t6 = BashOperator(
    task_id='t6',
    bash_command='sleep 2',
)
```

Q2.2 End to end stock volatility pipeline (20 pts)

- Now that you have understood pipelines in Airflow, design an end-to-end data pipeline that tracks a custom stock watchlist and flags high-volatility days.
- You have total freedom to pick the companies/cryptos you want to track and set your own alert thresholds.

